Rule-Based Semi Automatic Web Services Composition

Ehtesham Zahoor, Olivier Perrin, Claude Godart

Rule-based semi automatic Web services composition

Ehtesham Zahoor, Olivier Perrin and Claude Godart
LORIA, INRIA Nancy Grand Est Campus Scientifique
BP 239 54506 Vandoeuvre-lès-Nancy Cedex, France
{ehtesham.zahoor, olivier.perrin, claude.godart}@loria.fr

Abstract

In this paper we propose a rule-based approach for the semi-automatic Web services composition problem, giving end-user the control to guide the overall composition process. The end-user builds the composition flow by selecting known Web service instances or constrained Web service types, called nodes, and by connecting them using a set of control/data flow connectors. The specified nodes will then be bound to concrete Web service instances using a set of rule-based queries satisfying the associated constraints. When compared to the traditional approaches, our model is declarative, allows for specifying both the functional and non-functional requirements, provides connectors that include both the data and control flow aspects and aims to choose the one best matched Web service for a node instantiation.

1. Introduction

Traditional methods for Web services composition problem such as WSBPEL or WS-CDL have resolved the basic interoperability problem. However, in complex service oriented applications the services may need to be composed on the fly, with service instances not known in advance, and thus introducing the problem to automatically synthesize and adapt Web services composition process satisfying user request. There have been many approaches to the automatic composition problem but it is still considered highly complex task due to the rapid increase in the number of available services to choose from, the heterogeneity of the access protocols used and data formats they offer. Further, the proliferation of Web services may lead to a situation where we have many services with similar functionalities and thus we need to cater for the non-functional properties.

In this paper we propose a rule-based approach for Web services composition problem, putting the user in the control of the composition process. This leads to the semi-automatic Web services composition problem as the composition flow is guided by the user. The motivation for our work comes from the growing usage of mashups that are defined in the literature [3, 16, 1] as the new wave for composing Web services.

Our model allows user to select Web service instances and Web service types, called *nodes*, and connect them using the proposed set of connectors in order to define a process. When compared to the traditional approaches, our model has many differences: first, constraints on nodes include both the functional and non-functional specifications to be used for Web service discovery. Then, as mashups are the application level service composition and focus on composing the data from Web services [16], our proposal aims at providing a set of connectors that include both the data and control flow aspects. Further, in contrast to the procedural approaches, we propose a declarative approach to model the composition process and as opposed to the AI planning based approaches, we propose to select one best matched service (based on user specified criteria) as a result of node instantiation, this would be of critical importance due to the rapid increase in the number of available services. Our approach also handles the case when the service instantiation needs to be backtracked based on dependency between services and allows for propagation of newly chosen solution.

The paper is organized as follows. We discuss related work in section-2 and present the motivating example in section-3. We broadly discuss our proposal in section-4, while we detail the composition flow in section-5 and the concrete composition process in section-6. Implementation details are given in section-7 while section-8 concludes.

2. Related work

Web services composition is a highly active and widely studied research direction and there have been many approaches to automate the composition process. Most of these approaches can be divided into Workflow composition and AI planning based approaches, as discussed in [9]. The composition result can be regarded as a wokflow because it includes the atomic Web services and the control

and data flow between them. *Static* workflow composition approaches require an abstract composition flow to be specified and the selection and binding is performed automatically by the Web services composition process, while the *dynamic* workflow composition approaches require to both build the composition flow and select atomic service automatically based on user request as proposed in [11]. The composition process can also be regarded as a AI planning problem by assuming that each Web service can be specified by its pre-conditions and effects. These approaches require the user to specify the process by a set of pre-conditions and effects and the AI planners can then generate a plan without any pre-defined workflow. These approaches are based on: situation calculus [7], rule-based planning [5], theorem proving [15] and other approaches including [12].

The problem of traditional approaches (such as WS-BPEL or WS-CDL) is that all what is not explicitly modeled is forbidden. In fact, WSBPEL and WS-CDL have in common that they are highly procedural, i.e., after the execution of a given activity the next activities are scheduled. Seen from the viewpoint of an execution language their procedural nature is not a problem [14]. However, unlike a classical system, Web services tend to be rather autonomous and an important challenge is that all parties involved need to agree on an overall global process. Moreover, this way of modeling renders difficult to model complex orchestrations, i.e. those in which we need to express not only functional but also non-functional requirements such as cardinality constraints (one or more execution), temporal constraints, existence constraints, negative relationships between services, or security requirements on services (e.g. separation of duties). With current approaches, the designer should explicitly enumerate all the possible interactions and must either over-constrain or over-specify the orchestration. In case of multiple constraints (security and temporal for instance), it becomes very difficult to do that without declarative specifications. A more detailed discussion can be found in [8].

Our approach can be categorized as a composition framework (similar to the Astro approach [4]) and it provides the ability to dynamically discover the Web services by using the "service selection rule" to be processed. Our approach can also be considered as an extension to [6] but in this work we provide an extended set of connectors (to handle both control and data flow), introduce specification of non-functional properties, introduce concepts such as propagation and propose that some part of the composition may be static (with known Web service instances) while some other may require dynamic binding. Further, we argue the existence of private constraints for a service as proposed in [6]. Our approach proposes inferring the *worksWith* dependency using the service composability rules [5]. In addition, we have also proposed an implementation framework for our proposal using known Web service standards.

3. Motivating example

For the motivating example, we have chosen the case study of a SOA based Corporate Cash Management (CCM) solution[1]. Pierre has been hired as the treasury in an organization, the *bigCo*. One of his core responsibilities includes the CCM process, which is defined to be the process of managing a company's short-term resources, gathered for example from various financial institutions and ERP systems, to sustain its ongoing activities and to mobilize funds.

At *bigCo*, CCM is currently achieved by manually contacting different participating entities resulting in a tedious, less efficient and time consuming process. Pierre is willing to enhance this manual process of CCM by using the SOA principles, given that various financial institutions and internal systems being used at *bigCo* are exposing their functionalities using Web services.

In this context his responsibilities may include to compose cash positions held in multiple banks and internal ERP systems using the provided Web services; to determine investment or loan plans using the intelligent decision system Web service; to discover possible loan/investment options from various financial institutions based on some specified constraints (using the provided Web services); and to identify and execute Web services to get offer rating for the proposed investment or loan offers.

4. Proposed framework

Our proposal aims to provide a declarative framework for addressing the semi-automatic Web services composition problem, such as the one presented in the motivating example. In this section we will briefly discuss the main concepts related to our approach and will detail them in the sections to follow. The composition process starts when the

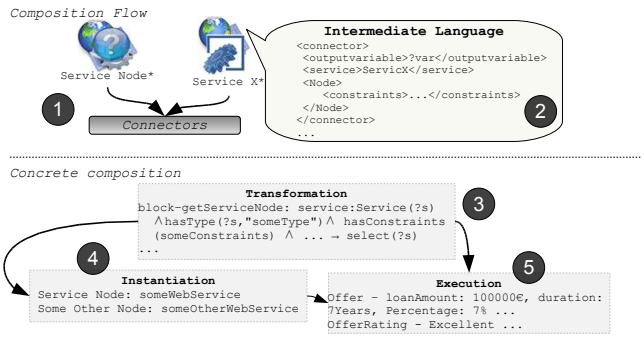

Figure 1. Proposed model components

user specifies the composition flow using a user friendly interface (see figure 1-1), allowing her/him to drag and drop

[1] http://www.oasis-open.org/presentations/security/KCronin.ppt

components as similar to the Mashup creation tools such as Yahoo! Pipes. The user interface will be backed up by a declarative language, called the intermediate language, (see figure 1-2), though the end-user does not need to know the details of the intermediate language and works directly at the interface level. The various concepts introduced in the declarative language (that map directly to the components provided on the user-interface) include:

- *Web services* - The user can specify the Web services instances known in advance.
- *Nodes* - If the Web service instance is not known in advance, the user can specify the Web service node which has a unique type such as Bank or Credit Union. User can also add constraints to the Web service node to be satisfied when binding the node to some Web service instance of the same type.
- *Connectors* - Connectors link different nodes and Web service instances and specify the control and data flow between them.

The intermediate language will then be processed by the composition engine to transform it into a set of rule-based queries, called *blocks* (see figure 1-3).

Then, the instantiation process will bound the Web service nodes to the actual Web services (see figure 1-4). Finally, the execution of query blocks will return the results (see figure 1-5).

5. Composition flow

5.1 Connectors

Connectors link different Web service nodes or instances. Below we highlight some of the basic connectors, including a special form of data flow connectors known as operators.

Sequence is the most basic connector that executes the next Web service in sequence. It specifies both the invocation of some Web service specified in the target variable and also stores the invocation result in the output variable. In its basic form, sequence is a control flow connector but we can augment it to handle the data flow by specifying the data to be passed as the input variable, which is then forwarded to the target Web service for some processing. The role of input variable here is more than just specifying the input parameters of the service to invoke, by considering that the input may have been obtained by some processing earlier by some other service and/or connector (and thus the data flow aspect). Sequence connector takes the following form, the *?inputVariable* is optional:

```
<sequence>
 <outputVariable>?resultVar</outputVariable>
 <inputVariable>?sourceVariable</inputVariable>
 <target>serviceName/node</target>
</sequence>
```

The **Split** connector specifies the control (and data) split to multiple *splitBlocks*. The split decision can be based on contents of input data and this special case is called *content-based-split*. Further, actual split can take one of three forms: same input data (or control) is forwarded to all *splitBlock*s, the *AND-Split*; to at-least one of the *splitBlock*s, *OR-Split* and to exactly one of the *splitBlock*s, *XOR-Split*:

```
<split scheme="AND/OR/XOR">
 <inputVariable>?sourceVariable</inputVariable>
  <splitBlock>
   <condition>some condition</condition>
   ...
  </splitBlock>
  <!-- other splitBlocks -->
</split>
```

The split *condition* is optional and is needed in the case of content based split. Further, the split connector has no output variable to bound the results. The separation between split and aggregation is guided by the different split and aggregation schemes and to handle this, we introduce the aggregate connector below.

The **Aggregate** connector receives a collection of results, for example from different *splitBlock*s or output variables. Once a complete set of results has been received, it binds a single aggregated result to the output variable. In order to decide that a set is complete, we introduce the following aggregation schemes: *all* - all the results should be considered, *exactly-one* - one of the results should be considered, *at-least-one* - the first result that is received is considered, and *subset* - only a subset of results is aggregated. However, when the aggregate connector is coupled with the XOR based split connector, the only possible aggregation scheme is *exactly-one*.

```
<aggregate scheme="(all/exactlyOne/at-least-one/subset)">
 <outputVariable>?resultVar</outputVariable>
 <!-- some result variables or splitBlocks -->
</aggregate>
```

Data operators These operators provide operations such as data transformations and validations for the data to be passed between different components. This includes a large variety of data transformation operations ranging from string manipulation, basic mathematical calculation operations to advance operations such as enriching data from service nodes and others. Some commonly used data transformation and validation operations such as *translator*, *enricher*, *filter*, *normalizer* and others are discussed in [16].

The proposed set of connectors can be compared to the OWL-S control constructs used for the composite process definition. A major difference is that our connectors can handle both control and data flows. When compared to the

sequence connector of OWL-S, our *sequence* connector is similar but it handles both control and data flows. For the split and aggregation process, two control constructs are introduced in OWL-S named *split* and *split+join*. Partial synchronization in OWL-S, i.e to split all and join some subbag, is similar to the proposed *split/aggregate* schemes. The *choice* OWL-S construct can be handled using the XOR based *split* connector while the *if-then-else* OWL-S control construct maps to the proposed *if-then-else* connector, which is not discussed due to space limitations. Further, the *iterator* connectors, similar to the proposed OWL-S *iterators* can be used, for example to process each of the results returned by some node instead of selecting the best match.

5.2 Constraints

For the nodes in composition flow, user can further add constraints that are not only to be satisfied during Web services discovery but also specify one specific path (solution) to choose from all available paths (solutions) for the Web services composition process. These constraints can be in the form of non-functional requirements such as security, reliability, quality requirements. They can also be in the form of some domain specific functional properties (loan duration, interest percentage for the motivating example).

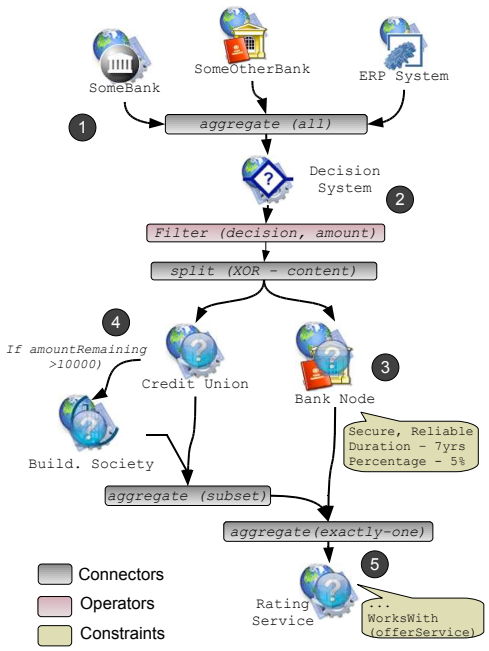

Figure 2. Motivating Example

The constraints can either be local to some node, can be based on properties related to a fragment of the process or based on the overall composition process. They can also be modified and passed between different nodes.

5.3 Example

Let us now review the motivating example and see how the composition flow can be specified. For the CCM scenario, the first part of composition requires Pierre to identify the current cash positions for the *bigCo*. As the Web service instances are known in advance, Pierre specifies the concrete instances for *someBank*, *someOtherBank* and for the *ERP System* Web services.

He specifies the *AND-split* as the connector which splits the user request to multiple Web service instances and then specifies the *aggregate-all* to get the results (see figure 2-1, for simplicity we have not shown the split operation of user request). The generated intermediate language code is shown below:

```
<aggregate scheme="all">
 <outputVariable>?assets</outputVariable>
 <split scheme="AND">
   <!-- splitBlocks with sequence to Web services -->
 </split>
</aggregate>
```

The aggregated information from banks and ERP system, i.e the `?assets` variable, is then sequenced to the intelligent decision making system to either opt for investment or loan option (see figure 2-2, sequence is the default connector and is represented by an arrow).

```
<sequence>
  <outputVariable>?DSResult</outputVariable>
  <inputVariable>?assets</inputVariable>
  <service>decisionSystemService</service>
</sequence>
```

The information returned by the decision system will be then filtered to get only the relevant data (see figure 2-2), space limitation restrict us to discuss the generated intermediate language code. We assume that the filtered decision and amount information are bound to `?decision` and `?invLoanAmount` variables respectively.

The filtered information will then be splited using the XOR split scheme to different financial institutions for possible investment or loan offers. If the decision is to get loan, information is sent to the first *splitBlock* of the generated intermediate language code shown below. As the bank Web services providing loan offers are not known in advance, Pierre specifies the *Web service node* of type Bank and provides constraints such as loan period, expected interest percentage and others (see figure 2-3).

```
<aggregate scheme="exactlyOne"> <!--Loan or Investment-->
 <outputVariable>?offer</outputVariable>
 <split scheme="XOR">
 <inputVariable>?decision</inputVariable>
 <splitBlock> <!--Loan decision -->
  <condition>?decision=Loan</condition>
  <sequence>
   <outputVariable>?bankLoanOffer</outputVariable>
   <inputVariable>?invLoanAmount</inputVariable>
    <node>
     <nodeType>Bank</nodeType>
```

```
     <constraints>
      <securityRating>High</securityRating>
      <reliabilityRating>High</reliabilityRating>
      <loanDuration>7Years</loanDuration>
      <interestPercentage>5</interestPercentage>
     </constraints>
    </node>
   </sequence>
 </splitBlock>
```

For the investment decision, let us further consider that the company wants to invest with a specific priority level; the information is sent to the second *splitBlock* which will first sequence the data to the credit union service node and in case of no complete offer, some building society service can be contacted (see figure 2-4). Information returned will then be aggregated by using the aggregate-subset connector. For the credit union and building society nodes, user can also specify constraints such as investment period, and some other non-functional properties (omitted due to the space limitations).

```
<splitBlock><!--Investment decision -->
 <condition>?decision=Investment</condition>
 <sequence>
  <outputVariable>?CUOffer</outputVariable>
  <inputVariable>?invLoanAmount</inputVariable>
  <!-- credit union node  -->
 </sequence>
 <operator name="filter">
  <outputVariable>?CUamount</outputVariable>
  <inputVariable>?CUOffer</inputVariable>
  <messagePart>offerAmount</messagePart>
 </operator>
 <operator name="subtract">
  <outputVariable>?remainingAmount</outputVariable>
  <!-- expression: ?invLoanAmount - ?CUamount -->
 </operator>
 <if>
  <condition>?remainingAmount>=10000</condition>
  <sequence>
   <outputVariable>?BSOffer</outputVariable>
   <inputVariable>?remainingAmount</inputVariable>
   <!-- building society node  -->
  </sequence>
 </if>
 <aggregate scheme="subset">
   <outputVariable>?investOffer</outputVariable>
   <inputVariable>?CUOffer</inputVariable>
   <inputVariable>?BSOffer</inputVariable>
 </aggregate>
</splitBlock>
 ...
```

Finally the investment or loan offer is filtered and sent to the rating service, which provides rating for the financial institution providing the offer. Again, as the service instance is not known in advance, user specifies Web service node as of type Rating. User can also add a special constraint *worksWith*, which will allow to filter only the rating services that "work with" the offer service selected earlier (see figure 2-5). We will discuss this special construct in section-6. Space limitations restrict us to discuss the generated intermediate language code.

6. Concrete composition

The generated intermediate language can then be used for the concrete composition process, which is divided into following three phases.

6.1 Translation

The concrete composition process starts by converting the intermediate language into a collection of rule-based queries, called *blocks* (see figure 1-3). These blocks act as the privacy control constructs as only the exposed results of some block will be available to the later blocks; whereas the unexposed internal details such as the constraints used by the block, will be hidden from other blocks. For each block, the associated query may have some local variables to operate on and possibly some constraints to be satisfied. It exposes its results by binding the output variables which are then available in the blocks to follow.

6.2 Instantiation

The translation process will be followed by the instantiation of all the Web service nodes specified by the user to some actual Web service instances (see figure 1-4). The instantiation process may however decide to delay the instantiation of the nodes whose invocation is conditional as an attempt to improve performance. The instantiation query block will operate on the knowledge base containing Web service instances with associated properties and will first filter only the Web services of the specified node type. Then, it will further filter the Web services based on the user-specified constraints. The result may be a collection of Web service and in case of a loosely constrained node, the result set can be very large. Our proposal thus aims to choose the best matched Web service based on some user-specified criteria such as the quality rating for the Web service, by assuming that some trusted third-party has quality ratings assigned to services. This choice can also be based on some other non-functional requirements or as our proposal aims to put user in the control of composition process, the user can also manually select the Web service.

If the instantiation result set for a node is empty then we have following possibilities. If some constraint is unsatisfied, user can be given option if she/he wants to relax the constraint. For example the user can decide to relax the quality rating from high to some other level in an attempt to discover new instances. Further, if the *worksWith* relation is unsatisfied, we need to backtrack to the results of dependent block to select some other instantiation solution and then proceed to finding solution for the current block. The process continues until all backtrack solutions have been explored. Finally when none of above two situations hold, the

composition process fails with notifying the user of the intermediate results and unbound node.

6.2.1 The *worksWith* dependency

We consider that a service *worksWith* some other service using the modified form of the composability rules discussed in [7]. These rules consider the syntactic and semantic properties of Web services. Syntactic rules include the rules for operation modes (one-way, request-response...), and the rules for binding protocols and data formats of interacting services. However, we can relax the rule for data formats by considering that two services providing data in different formats can still work with each other, by using the *translate* transformation operation as discussed earlier in section 5-1.

The Semantic rules include the *message composability* rule which defines that two Web services are composable only if the output message of one service is compatible with the input message of another service. In case of semantic Web services described using OWL-S, it is important to consider that the input and output parameters are defined in the domain ontology as specifying them as datatypes add very little to semantics ([10] has a detailed discussion). Further, the *operation semantic composability* defines the compatibility between the domains, categories and purposes of two services while the *qualitative composability* defines the requester's preferences regarding the quality of operations for the composite service. Then, the *composition soundness* considers whether a composition of services is reasonable, see [7] for details.

6.2.2 Backtracking

The backtracking process involves finding an alternative to some previously chosen node instantiation solution. Backtracking is needed when the *worksWith* relation for some node is unsatisfied resulting in empty result set.

6.2.3 Propagation

Once the backtracking process execution terminates, resulting in a newly chosen solution (instance), the composition solution must be recomputed and may require the propagation of newly chosen solution. This would likely be the case when a (partial) solution to the composition process has already been determined and backtracking to some higher node (in hierarchal order) may result in propagating the new solution. Further, propagation may also be needed when the user fine tunes the solution by manually selecting some other Web service after the instantiation process.

The propagation process will require recomputing the composition and this may result in significant overhead to re-instantiate the service nodes. Our proposal aims to re-instantiate only the Web service nodes that have dependancy on the node to backtrack (either a *worksWith* dependancy or a data dependancy).

6.3 Execution

The instantiation phase will be followed by the execution phase, which will return the results of the Web services composition (figure 1-5).

7. Implementation details

7.1 Architecture

In order to test our proposal, we have implemented a Java based application and tested it with multiple examples, including the motivating example. Below we briefly discuss various implementation related concepts.

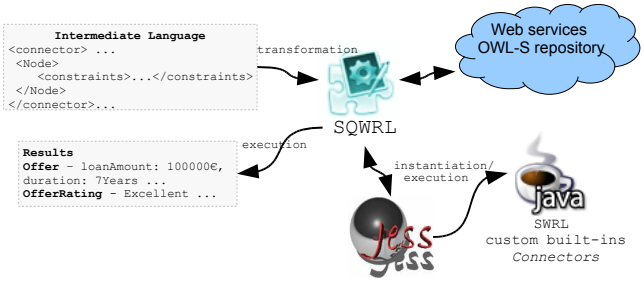

Figure 3. Implementation Architecture

OWL-S is a OWL-based Web service ontology providing constructs for describing a Web service in terms of a service profile (which describes what a service provides and allows service classification using *ServiceCategory* attribute and specification of non-functional properties using *ServiceParameter* attribute), the process model (which describes how the service works) and the service grounding (which specifies concrete details such as message formats, network addresses used and others).

The Semantic Web Rule Language (**SWRL**) is intended to be the rule language for the semantic web. The SWRL rules are written in terms of OWL classes, properties and individuals. SWRL also provides a set of core built-ins for strings manipulation, basic mathematical operations... It also allows to extend the core built-ins to add user defined built-ins. An example of a SWRL rule to express that a person with a older female sibling has a older sister can be written as:

```
Person(?p) ∧ hasAge(?p, ?pAge) ∧ hasSibling(?p,?s)  ∧
Woman(?s) ∧ hasAge(?s, ?sAge) ∧
swrlb:greaterThan(?sAge, ?pAge) → hasOlderSister(?p,?s)
```

In the rule above, *Person* is a class with a sub-class named *Woman* and *hasSibling* and *hasOlderSister* are OWL properties with domain and range of the class *Person*. The rule also uses the *hasAge* property (with domain as *Person* and range of primitive datatype *Integer*) and the SWRL builtin (*swrlb:greaterThan*) to add *hasOlderSister* property to all individuals who have older female siblings. The Semantic Query-Enhanced Web Rule Language (**SQWRL**) adds querying capabilities to SWRL by providing primitives to select, count and perform other operations on the results of a SWRL rule. Finally, SQWRL queries (and so as SWRL rules) require a rule-solver and for that we have used the **JESS** rule-solver (see figure-3 for the overall implementation architecture).

7.2 Example

For the implementation, we have programmed Java based Web services to return sample data form different systems. We have also created their OWL-S descriptions which include the specification of the non-functional properties using the approach specified in DAML[2] OWL-S examples. However, our proposal can also adapt the various QoS extensions to the OWL-S such as QoS-MO [13], QoSOnt [2] and other approaches that extend OWL-S for specifying QoS properties. For the CCM scenario, the concrete composition process starts with the translation of intermediate language code into a set of query blocks. Below we discuss the query blocks for various parts of the composition, for simplicity we will not discuss the node instantiation process separately and will highlight the instantiation blocks as they are encountered in the control flow of the composition process.

The first part of the composition (see figure 2-1), is static and although it does not require querying the rule base to search for Web services and invocation using rule engine; we here provide implementation of how such approach can be used.

```
service:Service(SomeBankService) ∧  service:supports(
SomeBankService,?SBSGrounding) ∧  grounding:hasAtomic
ProcessGrounding(?SBSGrounding, ?SBSAtProcGrnding)∧
grounding:wsdlDocument(?SBSAtProcGrnding,?SBSWsdl)∧
service:Service(SomeOtherBankService)∧...
connectors:aggregateAll(?assets,?SBSWsdl,
?SOBSWsdl,?ERPSWsdl) → sqwrl:select(?assets)
```

We have created a SQWRL query which searches through the OWL-S service model to get the grounding information for the Web services (we have only presented the query for *SomeBankService*) and then passes the URI's to a custom SWRL built-in, *aggregateALL*. This custom built-in then calls the Web services to get results and aggregates/bounds them to the output variable, `?assets`. We can then sequence the variable `?assets` to the *decisionSystemService* again using the custom SWRL built-in *sequence*. The result of the operation will be available in the variable `?DSResult`. We need to then filter the relevant message parts, such as decision and amount, from the result.

```
connectors:sequence(?DSResult,?assets,"decisionSystem
Service") ∧ connectors:filter( ?InvLoanAmount,
?DSResult,"Amount") ∧ connectors:filter(...)
→ sqwrl:select(?decision, ?InvLoanAmount)
```

Next the contents of the `?decision` variable, investment or loan decision, will be used to either opt for the loan or investment offer. Let us first consider the case that the loan offer decision is chosen. According to the composition flow, the loan request is sent to the bank service, not known in advance and which should be selected based on some constraints.

```
service:Service(?loanBank) ∧ service:presents(?loanBank,
?lbProfile) ∧ profile:serviceCategory(? lbProfile,
?lbCategory) ∧ profile:code(? lbCategory, ?categoryCode)
∧ swrlb:equal(?categoryCode, "522110") ∧

profile:serviceParameter(?lbProfile,?SPReliability)∧
profile:sParameter(?SPReliability, HighlyReliable) ∧
profile:serviceParameter(?lbProfile, ?SPDuration) ∧
profile:sParameter(?SPDuration, 7_Years) ∧ ...
other constraints ∧
service:supports(?loanBank, ?lbGrounding) ∧ grounding:ha
sAtomicProcessGrounding(?lbGrounding,?lbAtProcGrnding)∧
grounding:wsdlDocument (?lbAtProcGrnding, ?loanBankWsdl)

 → sqwrl:select(?loanBank, ?loanBankWsdl)
```

The query first selects the loan bank services based on the Web service classification code specified in the *serviceCategory* attribute of the service *profile*. We have used NAICS[3] categorization for the Web services and the code value 522110 is for commercial banking. So the initial part of the query will select only the bank services. Next we specify the non-functional constraints such as reliability and some functional properties such as the loan duration and interest percentage. Finally, in the last part of query we get the selected Web service wsdl URI from its grounding. As our approach proposes to select only one best match service, for the implementation we select the first Web service instance we get after executing the query.

Running the above query on our services repository will return two Web service instances, *someLoanBankService* and *someOtherLoanBankService* and we will select the *someLoanBankService* to be used. Next, we get the loan offer from the *someLoanBankService* using the sequence operation, space limitation restrict to discuss the query. Finally, we send the offer we have received from the *someLoanBankService* to the rating service, and again as the Web service is not known in advance, we specify the constraints and use query to search for the rating service as below.

[2]http://www.daml.org/services/owl-s/1.1/examples.html

[3]http://www.census.gov/eos/www/naics/

```
service:Service(?ratingService) ∧ service:presents(?....
∧ profile:code(?rsCategory, ?categoryCode) ∧
swrlb:equal(?categoryCode, "561450") ∧ ... some
constraints ... ∧ worksWith(?loanBank, ?ratingService)∧
 →  sqwrl:select(?ratingService, ?rsWsdl)
```

The query is similar to the one we mentioned earlier for the bank service, the NAICS code "561450" is for the rating services. In the constraints, we introduce the property *worksWith*, i.e we want to only search for the rating services that actually "work with" the *someLoanBankService*. For the implementation we have created a OWL-S property named *worksWith* that specifies which bank, credit union and building society services work with which rating service. This property can be handled dynamically by adding SWRL rules for service composability rules discussed earlier. The selected *someLoanBankService*, does not work with any rating service and the result set of the above mentioned query will be empty. Thus we need to backtrack and select some other service from the results of previous query, that is the *someOtherLoanBankService* and then search for the rating services that work with it. This time we find the *someOtherRatingService* and next we get the offer rating from the service using the sequence operation query (as similar to the *decisionSystemService* query).

Let us also consider the case when investment decision is chosen by the decision system, as similar to the queries mentioned above; we first get the credit union service (using the NAICS code "522130" for credit unions) and the corresponding offer by the selected service. Further, let us consider that the selected Web service does not provide the investment option for the complete investment amount and thus some building society service must be contacted for the remaining amount. The query below handles this, assuming the credit union offer amount is bound to the variable `?cuAmount` by some earlier query.

```
service:Service(?buildingSociety)∧service:presents
(?buildingSociety, ?bsProfile) ∧ .... swrlb:equal
(?bsCategoryCode, "522294") ∧... swrlb:subtract
(?amntRemaining,?InvLoanAmount,?cuAmount)∧
swrlb:greaterThanOrEqual (?amntRemaining, 10000)
→ sqwrl:select(?buildingSociety,?bsWsdl,?amntRemaining)
```

The above query only selects the building society service is the amount remaining, obtained after subtracting the credit union offer amount from the total amount, is greater than 10000. This serves as an example of how constraints can be modified and passed between nodes.

8. Conclusion

In this paper we have proposed a rule-based declarative framework for the semi-automatic Web services composition problem. Our approach requires the end-user to build the composition flow by specifying the Web service instances/types and data/control flow connectors to link them. The user interface is backed up with a declarative language, called intermediated language which is then used to instantiate the concrete composition process which involves translating the declarative language to rule based queries, called blocks, instantiating the Web service types and finally executing the process to get the results. We have also presented a sample Corporate Cash Management (CCM) scenario, that highlights our approach. A sample implementation to the sample scenario has been provided to discuss how our approach can be realized using the known Web services standards.

References

[1] D. Benslimane, S. Dustdar, and A. P. Sheth. Services mashups: The new generation of web applications. *IEEE Internet Computing*, 12(5):13–15, 2008.

[2] G. Dobson, R. Lock, and I. Sommerville. Qosont: a qos ontology for service-centric systems. In *EUROMICRO-SEAA*, pages 80–87, 2005.

[3] X. Liu, Y. Hui, W. Sun, and H. Liang. Towards service composition based on mashup. In *IEEE SCW*, 2007.

[4] A. Marconi, M. Pistore, and P. Traverso. Automated composition of web services: the astro approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008.

[5] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *VLDB J.*, 12(4), 2003.

[6] E. Monfroy, O. Perrin, and C. Ringeissen. Dynamic web services provisioning with constraints. In *OTM Conferences (1)*, pages 26–43, 2008.

[7] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW*, pages 77–88, 2002.

[8] M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In *Business Process Management Workshops, Austria*, 2006.

[9] J. Rao and X. Su. A survey of automated web service composition methods. In *SWSWPC*, pages 43–54, 2004.

[10] D. Redavid, L. Iannone, T. R. Payne, and G. Semeraro. Owl-s atomic services composition with swrl rules. In *ISMIS*, pages 605–611, 2008.

[11] H. Schuster, D. Georgakopoulos, A. Cichocki, and D. Baker. Modeling and composing service-based and reference process-based multi-enterprise processes. In *CAiSE*, 2000.

[12] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. In *In Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, pages 17–24, 2002.

[13] G. F. Tondello and F. Siqueira. The qos-mo ontology for semantic qos modeling. In *SAC*, pages 2336–2340, 2008.

[14] W. M. P. van der Aalst and M. Pesic. Decserflow: Towards a truly declarative service flow language. In *The Role of Business Processes in Service Oriented Architectures*, 2006.

[15] R. J. Waldinger. Web agents cooperating deductively. In *FAABS*, pages 250–262, 2000.

[16] E. Zahoor, O. Perrin, and C. Godart. Mashup model and verification using mashup processing network. In *CollaborateCom2008*. ACM, 2008.